Java - Non Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionalities.

- The *static* modifier for creating class methods and variables.
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

The Static Modifier

Static Variables

The *static* keyword is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

Static Methods

The static keyword is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

Example

The static modifier is used to create class methods and variables, as in the following example −

Live Demo

```java
public class InstanceCounter {

   private static int numInstances = 0;

   protected static int getCount() {
      return numInstances;
   }

   private static void addInstance() {
      numInstances++;
   }

   InstanceCounter() {
      InstanceCounter.addInstance();
   }

   public static void main(String[] arguments) {
      System.out.println("Starting with " + InstanceCounter.getCount() + " 
instances");

      for (int i = 0; i < 500; ++i) {
         new InstanceCounter();
      }
      System.out.println("Created " + InstanceCounter.getCount() + " instances");
   }
}
```

This will produce the following result –

Output

```
Started with 0 instances
Created 500 instances
```

The Final Modifier

Final Variables

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

Example

```
public class Test {

   final int value = 10;


   // The following are examples of declaring constants:

   public static final int BOXWIDTH = 6;

   static final String TITLE = "Manager";


   public void changeValue() {

      value = 12;   // will give an error

   }

}
```

Final Methods

A final method cannot be overridden by any subclasses. As mentioned previously, the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example

You declare methods using the *final* modifier in the class declaration, as in the following example –

```
public class Test {

   public final void changeName() {

      // body of method

   }

}
```

Final Classes

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

Example

```
public final class Test {

   // body of class

}
```

The abstract Modifier

Abstract Class

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise, a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

Example

```
abstract class Caravan {

   private double price;

   private String model;
   private String year;

   public abstract void goFast();   // an abstract method

   public abstract void changeColor();

}
```

Abstract Methods

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.

If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

Example

```
public abstract class SuperClass {

   abstract void m();   // abstract method

}


class SubClass extends SuperClass {

   // implements the abstract method

   void m() {

      .........

   }

}
```

The Synchronized Modifier

The synchronized keyword used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

Example

```
public synchronized void showDetails() {

   .......

}
```

The Transient Modifier

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

Example

```
public transient int limit = 55;   // will not persist

public int b;   // will persist
```

The Volatile Modifier

The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

Example

```
public class MyRunnable implements Runnable {

   private volatile boolean active;



   public void run() {
      active = true;

      while (active) {   // line 1

         // some code here

      }

   }


   public void stop() {

      active = false;   // line 2

   }

}
```

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1, the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.